Calcul haute performance avec OpenCL sur GPU

Sebastien Curt

Noname manuscript No.
(will be inserted by the editor)

Calcul haute performance avec OpenCL sur GPU

Curt Sébastien
Maître de stage : Vincent Danjean

the date of receipt and acceptance should be inserted later

Table des matières

Curt Sébastien
2 chemin de tétraz, 73000 Jacob Bellecombette
E-mail: curt.sebastien@gmail.com

1 Introduction

De nos jours les machines possèdent de plus en plus de processeurs ou des matériels spécialisés dans l'exécution de calculs en parallèle. L'exécution de calculs en parallèle permet de diminuer le temps de calcul. L'équipe de recherche MOAIS s'intéresse au parallélisme de tache depuis plus d'une dizaine d'année. Le programme Xkaapi développé par cette équipe a été récemment amélioré pour permettre l'exécution de certaines tâches sur les processeurs des cartes graphiques (GPUs) qui seront étudiés dans un premier temps afin de montrer qu'ils sont adaptés au calculs parallèles.

Cependant ce programme nécessite d'utiliser l'environnement CUDA pour travailler. Cette environnement est limité aux du constructeur Nvidia. Nous allons montrer un moyen de programmer les GPUs avec OpenCL. Nous montrerons aussi dans le même temps les points communs existants entre CUDA et OpenCL.

Enfin nous verrons l'utilisation d'OpenCL pour le HPC avant de conclure.

2 Les CPU et les GPU

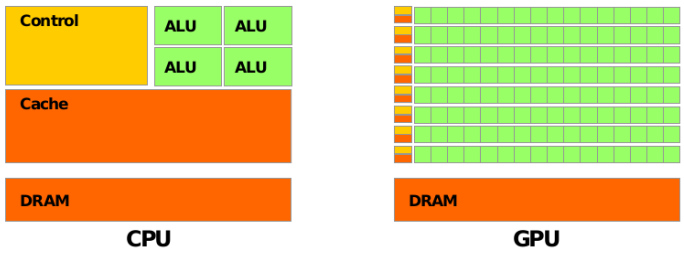

Figure 1 Architecture d'un CPU (gauche) et architecture d'un GPU (droite)[1]

Comme nous pouvons l'observer sur la figure 1 un CPU est composé d'un centre de contrôle, de peu d'*ALU* (*Arithmetic and logic unit*) qui sont les centres de calcul d'un ordinateur et de deux types de mémoire la *DRAM* (mémoire ou sont chargés les programmes) et d'un cache (mémoire à accès rapide) important.
De l'autre côté les GPU sont composés des mêmes éléments mais dans des proportions différentes. En effet nous observons un cache en quantité faible. L'espace ainsi libéré est remplacé par des *ALU*. Les CPU sont conçu pour diminuer le temps de latence entre différents calculs . La diminution du temps de latence peut être atteint en augmentant la quantité de cache. Le cache prend de la place sur un espace limité diminuant ainsi le nombre d'unité de calcul.

Ils sont utilisés pour du parallélisme de tache. De multiples taches sont mappés sur des threads multiples. Chaque tache exécutant différentes instructions, le changement de contexte entre chaque tache est une opération lourde en temps. De plus chaque thread doit être géré et ordonnancé de manière explicite, aboutissant à un débit de travail (*throughput*) lent.

Au contraire les GPU sont conçus pour supporter un temps de latence élevé. Cela permet de diminuer fortement la quantité de cache. Gagnant ainsi plus de place pour pouvoir mettre des unités de calcul, il en résulte que les GPU ont un débit de calcul nettement plus élevé qu'un CPU. Ainsi les GPU sont utilisés pour du parallélisme de donnée (*data parallelism*). Une seule instruction est exécuté sur plusieurs données en parallèle (modèle SIMD, Single Instruction Multiple Data). Les GPUs exécutent ainsi des centaines voir des milliers de threads léger en concurrence sur plusieurs centaines d'unités de calcul. De plus les threads sont gérés par par le matériel facilitant ainsi les changements de contexte faisant des GPU d'excellents candidats pour le calcul parallèle.

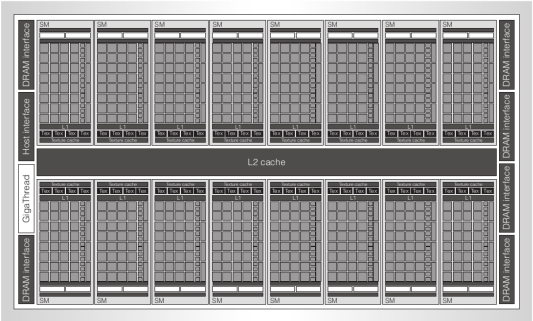

Figure 2 Architecture d'un GPU Fermi avec 512 processeurs CUDA organisés comme 16 flux de multiprocesseurs (SMs) partageant un deuxième niveau de cache (L2) en commun, avec 6 interface *DRAM* de 64-bit, et une interface hôte avec le CPU hôte, un système de mémoire, et des puces d'entrées sorties. Chaque flux multiprocesseurs possède 32 cœur CUDA

Les cartes CUDA[3] sont construites autour de tableaux de taille variable les *Streaming Multiporcessors multithreadés* (SMs). Le multiprocesseur crée, gère, ordonnance et exécute des threads par groupe de 32 threads parallèles appelés des warps. Chaque SM possède 32 cœur de processeur CUDA, 16 unités de lecture/écriture, 4 unités de fonctions spéciales, 64Ko de mémoire (L1) cache, 128Ko de banc de registre, un cache d'instruction et deux ordonnanceur de warp et une unité de distribution d'instruction. Les GPU cudas peuvent donc gérer beaucoup de thread en parallèle, confirmant ainsi l'utilité des GPU dans les calculs hautement parallèles.

Cependant les GPUs ne sont pas faciles à programmer. Nous allons voir un moyen de les programmer avec OpenCL.

3 Cuda et OpenCl

OpenCL est un framework pour la programmation parallèle qui inclue un langage, une API, des librairies et un système d'exécution pour le développement de logiciel. La programmation avec OpenCL s'articule autour de certains modèles. Les modèles sont les suivants : la plateforme, la mémoire, l'exécution et la programmation.

3.1 La plateforme :

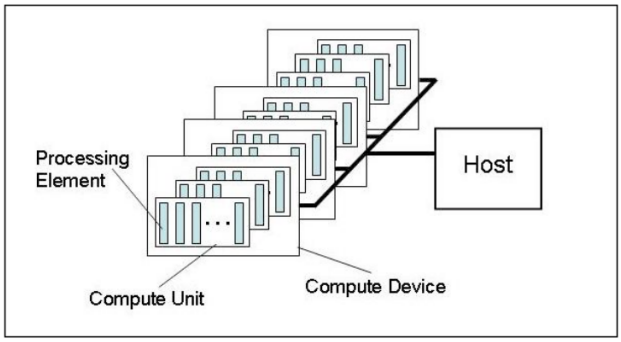

Figure 3 Modèle de plateforme

Une plateforme est un modèle hiérarchisé basé sur un *host* connecté à un ou plusieurs *device*. Un *device* est divisé en une ou plusieurs *compute unit* (CU) qui sont elles mêmes divisés en *processing unit* (PE). Les calculs se produisent à l'intérieur des ressources de calculs.

3.2 L'exécution :

Le modèle d'exécution de OpenCL est proche de celle de l'architecture de CUDA. L'exécution d'un programme OpenCL est divisé en deux parties, la première concernant les *Kernels* qui sont les programmes exécutés sur un ou plusieurs *devices* et la seconde, le programme *host* qui a pour but de définir le contexte pour les *kernels* et gère leur exécution. Le contexte pour l'exécution d'un *kernel* correspond à l'ensemble des *devices* utilisés, au *kernel* qui s'exécutera sur les *devices* au programme source et exécutable qui

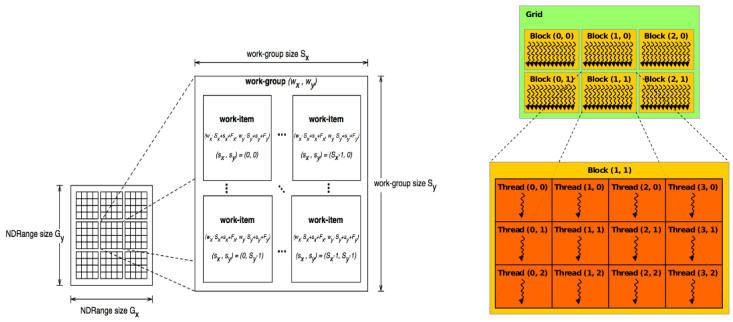

Figure 4 Modèle d'exécution d'OpenCL[8] (à gauche) et de Cuda[5] (à droite)

implémentent le *kernel* et aux objets mémoire qui contiennent les valeurs sur lequel un*kernels* va calculer. Il existe dans CUDA et OpenCL une hiérarchie dans les threads.

La figure 4 de la page 5 offre une vision globale du modèle d'exécution. Nous allons expliciter en parallèle les équivalences entre les deux modèles d'exécution.

Les threads : Chez CUDA il s'agit des *CUDA thread* et dans OpenCL ils se nomment les *work-items*. Un thread est un fil d'exécution. Chaque thread a un numéro d'identifiant unique.

Les blocs de threads : Chez CUDA ils sont nommés les *CUDA thread Blocks*, l'équivalent OpenCL sont les *work-groups*. Un bloc de thread contient un nombre limité de threads. Un bloc est assigné à un SM. Chaque Bloc possède un identifiant unique. À l'intérieur d'un bloc les threads peuvent être ordonnés dans un ensemble uni, bi, tri-dimensionnel de threads.

L'espace d'adressage total : L'espace d'adressage est l'ensemble de tous les blocs. Il peut représenter un ensemble à une, deux ou trois dimensions. Chez CUDA l'espace d'adressage total correspond à un *grid* alors que dans OpenCL il s'agit d'un *NDRange* où N est la dimension du système.

Il est important de savoir que Des GPUs concurrents peuvent exécuter un kernel à la fois. Ils ne peuvent pas supporter des taches parallèles entres les différents kernels. Cependant les GPUs supportent des taches parallèles à l'intérieur d'un kernel à partir du moment que tous les *works-items* suivent le même chemin d'exécution.

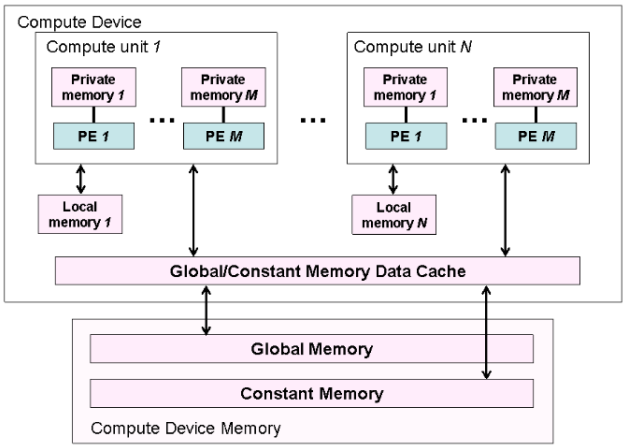

Figure 5 Modèle de mémoire d'OpenCL[9]

3.3 La mémoire :

Dans CUDA et dans OpenCL il existe une hiérarchie de la mémoire. Nous allons les présenter de la plus rapide à la moins rapide en temps d'accès et par équivalence.

La mémoire la plus petite correspond à la *local memory* et aux *registers* dans CUDA et à la *private memory* dans OpenCL. Il s'agit d'une mémoire qui est propre à un thread.

La mémoire au dessus est la *shared memory* dans CUDA et la *local memory* dans OpenCL. Il s'agit d'une mémoire visible (partagée) par tous les threads d'un seul bloc thread CUDA et tous les threads d'un seul *work-group* OpenCL.

Au dessus se trouve la *global memory*, *texture memory* dans CUDA et la *global memory* dans OpenCL. Ces mémoires sont partagées entre tous les blocs de thread de dans CUDA et les *work-groups* de OpenCL. Au même niveau se trouve aussi une mémoire accessible uniquement en lecture.

Tous ces niveaux de mémoire peuvent être vérifiées sur la figure 5 page 6 afin d'avoir une aide visuelle.

3.4 La programmation

Chez OpenCL on peut utiliser deux modèles de programmation. Le premier correspondant au *Data Parallel Programming*. Modèle dans lequel les données sont distribuées entre différents nœuds qui calculent en parallèle [7]. Le second correspondant au *task parallel programming* dans lequel on distribue l'exécution des processus à travers différents nœuds de calcul.

Contrairement a CUDA les *SIMT* (Simple Instruction Multiple Thread) sont non affichés dans OpenCL.

4 OpenCL et le HPC

L'équipe MOAIS s'intéresse au parallélisme de tâche. L'équipe a développé plusieurs environnement dont le dernier, XKaapi, permet de programmer une application avec des tâches de grain fin. Cet environnement a récemment été amélioré pour permettre l'exécution de certaines tâches sur les processeurs des cartes graphiques (GPGPU), permettant ainsi d'améliorer grandement les performances pour certaines classes de programmes. L'extension a été développé en CUDA. Donc XKaapi ne peut être déporté uniquement sur carte NVIDIA.

Dans le laboratoire des tests de recouvrement de transfert de données a déjà été réalisé sur carte graphique NVIDIA. Ce test mettait en jeux un calcul de multiplication de matrice par bloc implémenté en CUDA.

Nous voulons savoir s'il est possible de faire du recouvrement de transfert de données par du recouvrement de calcul en OpenCL et si oui quelle quantité de donnée de transféré est recouvert par du transfert de calcul. Ainsi nous pouvons faire des différences d'efficacité entre CUDA et OpenCL.

La réponse concernant le recouvrement du transfert de données par du calcul est oui. En effet dans la SDK de NVIDIA il existe un programme qui porte le nom « OpenCL Overlapped Copy/Compute Sample »qui test le pourcentage de recouvrement sur calcul d'hypoténuse d'un ensemble de triangle. Après test sur un ordinateur équipé d'un carte graphique NVIDIA GeForce GTX 580, carte graphique possédant une architecture de type FERMI. Le programme permettait de déterminer un recouvrement de donnée à hauteur de 30%.

Le modèle de programmation qui nous intéresse est celui du *Data Parallel Programming*. En effet nous allons devoir faire le même calcul sur différents sur une grande quantité de donnée. Une expérience a été mise en place. Cette expérience consiste à faire exécuter un *kernel* sur une carte graphique. Le programme hôte découpe en sous blocs des matrices de taille très importante. Nous enverrons sur la carte graphique les données pendant qu'elle calcule le résultat d'un produit de matrice d'un sous bloc précédemment envoyé. De plus on essayera de récupérer, en même temps que nous enverrons les données, le résultat du calcul précédant. Nous ferons varier la taille de la matrice, ainsi que la taille des sous blocs de matrice.

5 Conclusion

OpenCL est un framework proche du modèle de programmation de CUDA. Il a l'avantage d'être utilisable sur différentes plateforme (cartes graphiques AMD, CPUs...). A l'heure actuelle il n'y a pas de résultats d'expérience produites. Dans un premier temps il faudra terminer les tests. Nous permettrons de tester OpenCL sur différentes plateformes afin d'observer des disparités ou des convergences d'efficacité. Une fois analysés, les résultats permettrons aussi

de déterminer si OpenCL pourra être pris en compte pour un éventuellement être sélectionné pour un développement de XKaapi. Cela permettra ainsi à XKaapi d'être utilisé sur plusieurs types de plateformes. D'autant plus que OpenCL étant assez proches de l'intégration pourra se faire relativement facilement.

Références

1. NVIDIA. OpenCL programming guide for the CUDA architecture, [document électronique]. Version 2.3 (27/08/2009). Page 7. www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf
2. Ashu Rege, An Introduction to Modern GPU Architecture. [document électronique]. ftp ://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf
3. NVIDIA. cuda c programming guide. [document électronique], http ://docs.nvidia.com/cuda/cuda-c-programming-guide/ [page consultée le 13/06/2013]
4. Aaftab Munshi, The OpenCL Specification. [document électronique], version 1.2, revision 19 (14/11/2012). www.khronos.org/registry/cl/specs/opencl-1.2.pdf
5. NVIDIA. OpenCL programming guide for the CUDA architecture, [document électronique]. Version 2.3 (27/08/2009). Page 12. www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf
6. Intel. Intel® Xeon Phi™. Novembre 2012. Page 12.
7. Wikipedia. Data parallelism. [en ligne]. http ://en.wikipedia.org/wiki/Data_parallelism. [page consultée le 14 juin 2013]
8. Aaftab Munshi, The OpenCL Specification. [document électronique], version 1.2, revision 19 (14/11/2012). Page 25. www.khronos.org/registry/cl/specs/opencl-1.2.pdf
9. Aaftab Munshi, The OpenCL Specification. [document électronique], version 1.2, revision 19 (14/11/2012). Page 28. www.khronos.org/registry/cl/specs/opencl-1.2.pdf

Remerciement

Je remercie mon maître de stage, Vincent Danjean, pour m'avoir aidé pour ce stage et qui ma permis de découvrir de plus proche le monde de la recherche.

Je tiens aussi à remercier tous les gens qui m'ont entourés pendant la durée du stage.